The ActivePolygon Polygonal Algorithm for Haptic Force Generation

Tom Anderson
Nick Brown
Novint Technologies

Abstract

Algorithms for computing forces and associated surface deformations from a polygonal data set are given, which can be used to haptically and graphically display virtual objects with a single-point cursor. A culled collision detection algorithm is described that works in real-time with large data sets utilizing an oct-tree method. After a collision is detected, forces are created based on the local area near the cursor, keeping track of an active polygon. This creates a method that is effective and scalable for large models. The 'Bendable Polygon' algorithm for visual rendering of computer generated surfaces is also given.

Introduction

Many of the most common data formats for 3D computer-generated worlds utilize polygonal representations for objects. In order to take advantage of the large existing quantity of polygonal data sets, and a common standard in environments and hardware acceleration, a method for creating haptics forces based on polygons is necessary. The following paper describes an algorithm that can be used to create the forces on polygonal objects. The ActivePolygon algorithm was implemented with triangular datasets, however the concepts can apply to any type of polygonal dataset.

The algorithm first focuses on determining if the user point, or cursor, has touched an object, which requires a collision detection algorithm. A culled collision detection algorithm is described that works in real-time with large data sets. Then forces are created based on penetration depth and the relative position of the cursor to the object's facets. Graphically, the Bendable Polygon technique is described in which a local area of an object is broken up to allow for small-scale visual deformations. Larger scale deformations occur in the haptic domain through a system of springs and dampers.

Collision Detection

The first step in creating the forces for an object is to find if the cursor is touching the object. This means that as the cursor moves, collisions between the cursor and the object's facets must be checked. After a collision is detected, forces are then determined and presented to the user.

A simple way to do collision detection is to check if the cursor has moved through any of the polygons in an object. This can be accomplished by taking the line segment from the cursor's current and previous positions each loop of the cycle, and comparing that segment with every one of the polygons in an object. If the line segment intersects any of the polygons then a collision has occurred.

This can be extremely time consuming, however, if the object consists of many polygons. It is inefficient to check every one of the polygons in an object each cycle of the loop. The process can be sped up by pre-processing the data and culling the polygons that are not in the cursor's vicinity during run-time, which allows real-time collision detection even with large data sets. An oct-tree is used to subdivide the space around the object. During collision detection, each cycle of the haptic loop all leaf nodes which the cursor has moved through are determined (usually this is only the current leaf node) and only the polygons within these nodes are checked for collisions.

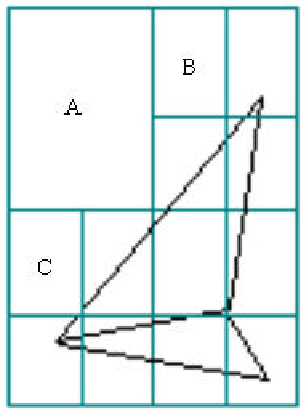

Figure 1: oct-tree based culling of a polygonal object.
Nodes A, B and C are empty, the remaining nodes contain one or both of the polygons.

To populate the oct-tree, first all the polygons are added to a single parent node. This node is then divided into eight octants all polygons within the parent are checked for overlapping each child octant, and added to the child's polygon list. Then, each of the octants is divided and it's polygon list transferred to the new children. An octant is not subdivided if contains a minimum number of polygons.

In pre-processing the oct-tree culling data, there are three situations in which a polygon should be included in an octant's checking domain as shown in Figure 2. The first is when any vertices of the polygon lie within the octant. The second occurs when any of the edges of the polygon intersect any of the sides of the octant's. The third situation occurs when any of the edges of the octant's intersect the polygon.

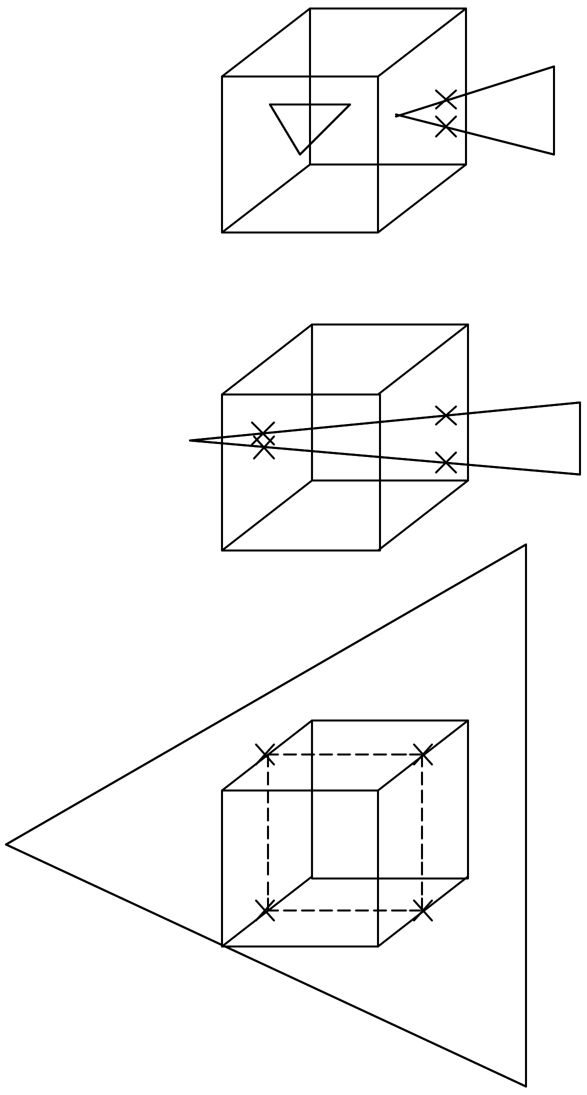

Figure 2: Voxel-Polygon events.
Top: 1 or more vertices in the voxel.
Middle: Polygon edge intersects a voxel side.
Bottom: Voxel edge intersects the polygon

If the tool's start position in the X direction is larger than its end position, octants are queried from right to left, otherwise from left to right. Setting the same checks in Y and Z ensures that the octant that will be collided with first is checked first.

The main consideration in using this type of culling comes from trade-offs in time and memory usage. The culling represents a method in which the object's size can grow to contain many millions of polygons, and the algorithm would still be able to do the collision detection in real-time. This would require very large amounts of memory, however. The maximum depth of the oct-tree and the maximum number of polygons within a leaf node can be changed to make a trade off between memory usage and processing time.

Force Generation

After a collision is detected, the forces must be presented. When the cursor touches an object, the specific polygon that was initially touched becomes the active polygon. The forces are in the normal direction and are proportional to the penetration depth into an object, which is measured from the currently active polygon. The direction of the force is interpolated at edges as the cursor moves across an object, and as the current polygon changes, to create a smooth feeling across facets. When the penetration depth of the active polygon becomes negative, the cursor has left the object, the forces are discontinued, and the collision detection algorithm is used again.

Normal Direction for Polygons

An initial issue is encountered because of the nature of a polygonal data set. The outward direction on a polygon is determined from the ordering of the points it contains. The direction that is considered outward is important for both graphics and haptics. In graphics, the outward direction is used to determine shading effects. In haptics, the outward direction is used in collision detection, force direction, and in interpolating between polygons.

A typical way to overcome this problem, which has become a standard in graphics, is to pre-process the points and order them so that all the vertex listings are consistent. If a data set is not already vertex-ordered correctly, then enclosed objects can be checked to make sure the outward direction remains consistent over a surface.

To find out if the vertices in any given polygon are ordered correctly, a point, point 'A', is projected from the center of a polygon in the normal direction of that polygon to a sphere that encloses all of the polygonal objects, to give point 'B' as shown in Figure 3.

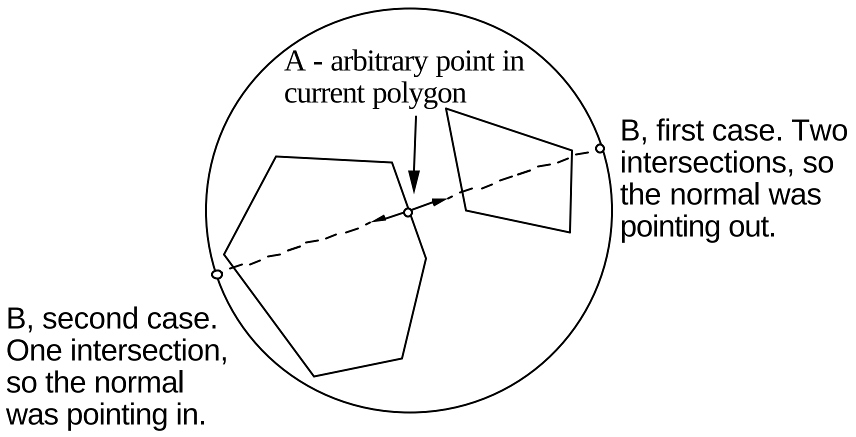
A - arbitrary point in current polygon
B, first case. Two intersections, so the normal was pointing out.
B, second case. One intersection, so the normal was pointing in.

**Figure 3: Finding if a polygon's vertices are correctly ordered.
The two cases of point B represent two different ordering conventions.**

Then the number of polygons that are intersected by the segment from point A to point B are counted. When objects are enclosed, an even number of intersections implies that the original normal was pointing outward and an odd number of intersections means that the normal was pointing inward.

Determining the Active Polygon and Penetration Depth

Once a collision is detected, one would then like to be able to slide the cursor from one polygon to another to touch the entire object. Several problems arise while trying to do this. First, the active or current polygon must always be known so that the direction of the force can be determined. Second, while sliding across polygons, if there is a sudden change in magnitude or direction of the normal force, then corners feel sharp and distinct even when there is only a slight angle between the adjoining polygons.

Originally, the transition from one polygon to another was accomplished by finding the distance from the cursor to the three edges of the polygon's normal projection. If the cursor crossed the projection then there would be a new active polygon. The problem with this approach was that the distance from the cursor to the current polygon’s surface would change when changing polygons, and a small jerk would be felt even when the normal direction was interpolated correctly as shown in Figure 4a. This is a problem not found in graphics interpolation because a second variable, depth, is included in the overall interpolation. Also, the distance to the edge of the plane would change, which is used in interpolating the direction of the force. And finally, there can be places within an object that are not in any polygon’s projection.

Therefore, a different method was determined in which the distances to ‘edge’ planes rather than the normal planes were found (Figure 4b). An edge plane, for a given edge, is determined from the two polygon vertices defining the edge and a vector that is the average of the normals of the polygons sharing the edge.

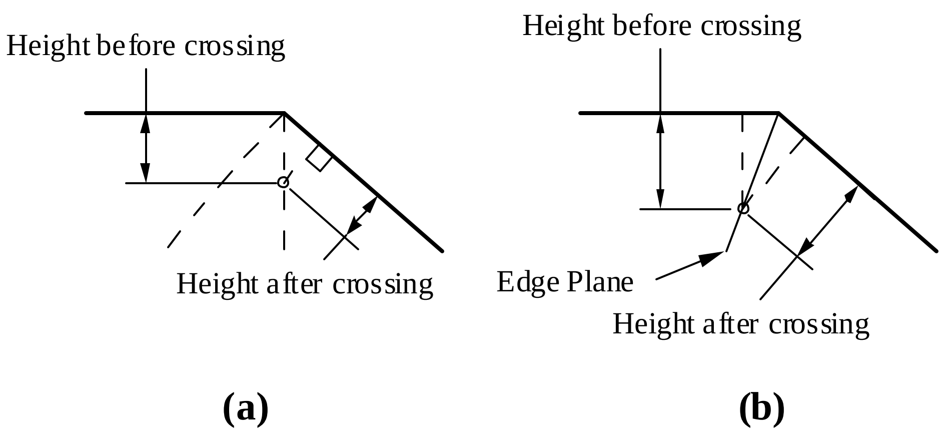

Figure 4: Determination of a change in the current polygon.

The method shown in Figure 4b makes the penetration depth and distance to the edge planes consistent while sliding to another polygon. When a change in the active polygon is detected, the new active polygon can be found by finding the other polygon that contains the two vertices in the edge plane that was crossed. This information can be preprocessed and stored in an array rather than finding it as the cycle runs, which saves on cycle time.

Force Direction Interpolation

In addition to consistent depth of penetration distances, the directions of the forces need to be consistent to keep the edges smooth. This is accomplished by interpolating between adjoining polygons in a way similar to Phong shading in graphics. When the projection of the cursor into the active polygon comes within a fixed distance from the edge planes, the normal direction is interpolated and then normalized.

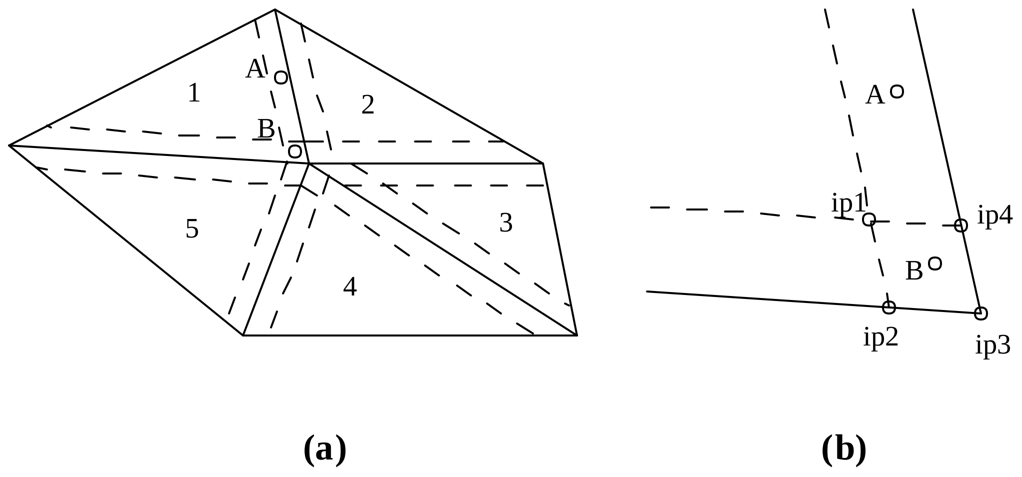

Figure 5: Interpolation of the normal direction.

The direction at point ‘A’ is interpolated continuously (not necessarily linearly) between polygons 1 and 2. At point ‘B’, the normal is interpolated from 4 points, ip1 through ip4, all of which are normalized. Ip1 is the normal direction of polygon 1. Ip2 is the average of the normals of polygons 1 and 5. Ip4 is the average of the normals of polygons 1 and 2. Finally, Ip3 is the average of the normals of all five polygons. Care must be taken to make sure the direction is continuous while interpolating over boundaries.

Overall, the interpolation over the edges of the polygons presents an interesting psychophysical effect. If the forces are interpolated correctly, then the shape of the object is perceived more from the direction of the forces than from the cursor’s actual position. For example, the facets of a polygonal sphere are easily distinguishable with no interpolation. With the introduction of interpolation, the facets become less noticeable and the sphere feels rounder. As the area over which each facet is interpolated increases, until the forces over the entire facet are interpolated, the sphere appears to ‘fill out’. Although a user is still feeling a faceted sphere, it appears completely round and smooth. This can be a powerful effect as the interpolation can be used to modify the way that an object’s shape is perceived. Additionally, if an edge is supposed to be distinct, then the interpolation can be turned off which will produce a sharp edge.

Acute Edges

If the tool is touching the outside of an acute edge the force will not be interpolated across that edge, which will make the edge feel distinct and sharp, rather than curved. The force is interpolated across the edge in Figure 6a. The force in Figure 6b is taken from the current polygon only. Additionally, there is an issue with acute edges known as the thin-wall problem. This is a common problem in haptic rendering algorithms where the cursor can unintentionally push though a thin wall, such as a knife blade. To handle this problem in the ActivePolygon algorithm, the edge plane that is normally used to transition to a new active polygon, as shown in figure 6a, is not computed or used for acute angles, as shown in Figure 6b.

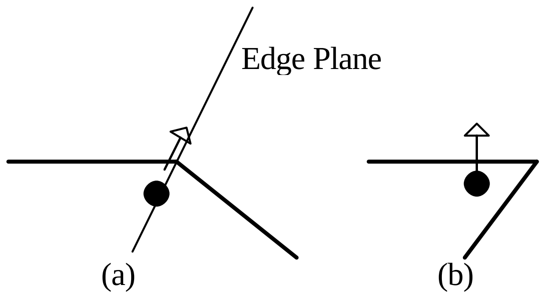

Figure 6: Interpolation force for an exterior acute edge.

If the tool is on the inside of a very acute edge it may travel some distance from the polygon before approaching the edge plane, which can create inconsistencies in the forces. Figure 7 shows a cursor point that has touched an interior edge of an acute angle, and has penetrated into the object.

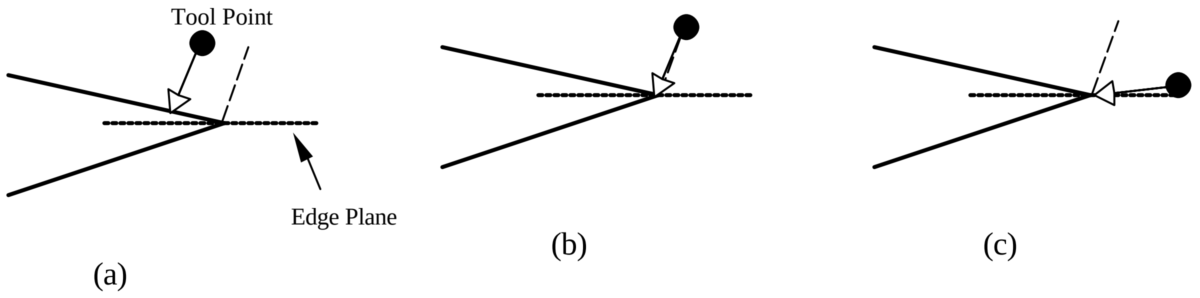

Figure 7: Interpolation force for an interior acute edge.

The point is within an orthogonal projection of the polygon in Figure 7a, so the force is in the polygon's normal direction. In Figure 7c, the tool has moved outside the polygon's orthogonal projection, so the force is directly towards the edge. At the point where the direction of the force changes from the normal direction of the polygon to the edge, as shown in figure 7b, both of those computations are equal, thus giving a consistent force. To increase the speed of the algorithm, acute edges are marked during pre-processing.

Three or More Polygons Sharing a Common Edge

As has been described, the ActivePolygon algorithm utilizes a local region when computing forces. The currently active polygon and the polygons sharing its vertices are used to create forces based on a single point. In order to speed up the algorithm, and reduce processing time, polygonal neighbors are preprocessed. However, when a single edge is shared by three or more polygons, special processing must be done. In this case, the neighbor information is dependent on the side of the polygon. When three or more polygons share a common edge, one side of a polygon has one neighbor, and the other side of the polygon has another neighbor. If an edge has only two polygons attached to it then the front and rear face neighbors are the same.

The side of the polygon which is currently active will determine which of the two neighbors is used for processing.

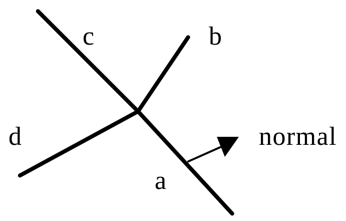

Figure 8: An edge with three neighbors.

Shown in Figure 8 is an edge that is shared by 4 polygons. Polygon a will have polygon b as a front neighbor and polygon d set as a back neighbor. Polygon c will not be a neighbor of a.

Bendable Polygon Algorithm

As the forces are presented to the user, the visual aspects of the virtual object must be consistent with the object. A compliant object should deform as the cursor moves into it. If the object does not deform, then the cursor can be lost visually within it. One way to solve this problem is to simply project the visual cursor, in the direction of the force, to the surface of the object.

However, if the object is very compliant, then this can create a discrepancy between the visual and haptic senses. To solve this, the Bendable Polygon technique is used. When the cursor first touches a polygon, it is split into 6 different polygons as shown in Figure 9. The cursor is projected normally to the plane of the active polygon, and then that point is projected to the edges of the polygon, making base points that are the framework for the approach. The ‘edge base points' move as the cursor moves, always normally projected to the sides of the current polygon. When the polygons are Gouraud or Phong shaded, the edges look smooth and the polygon seems to bend. The effects of the Bendable Polygon technique decrease with increased graphical detail, but for relatively large polygons, the effect works well and allows for lower levels of detail on a deformable object.

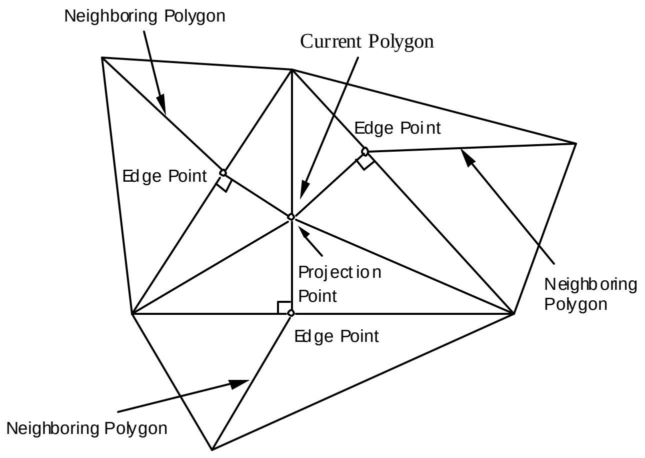

Figure 9: Base Points in the Bendable Polygon algorithm.

The cursor is then connected by a spring and a damper to each of the vertices in the active polygon and to each of the edge points. The cursor does not have any forces applied to it by these springs. All six of those points, in turn are connected to the base points shown in Figure 9, also by springs and dampers.

When the cursor moves into a polygon, the vertices (open circles) are pulled away from their respective base points (filled circles), making the polygon bend as shown in Figure 10.

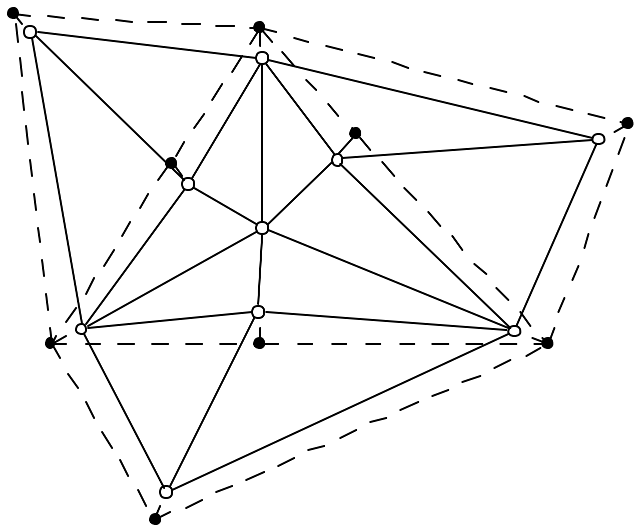
Figure 10: Base Points and Vertex Points in the Bendable Polygon algorithm.

The dashed lines represent the object while it is not deformed, and the solid lines represent the polygons that the user sees after it is deformed. As the cursor moves towards an edge, the springs from the edge points to their respective base points lose strength, so the indentation in a polygon remains consistent even as the cursor moves across different polygons. Different spring constants and different levels of strength reduction give different amounts of indentation (i.e. a small indentation on a water balloon as opposed to a larger indentation on a trampoline). The springs from the corner vertices to their base points work similarly, in that they lose strength as the cursor approaches them, so the indentation remains consistent at the corners of the polygons as well.

Then, each of the vertices throughout the object is connected by springs and dampers to each of the vertices touching it, to give an overall ability of large-scale deformation. In large data sets, the number of calculations would become extremely large, so springs might, for example, only be connected to vertices in the general surrounding area of the cursor, depending on the application, or a finite element analysis can determine the object's deformations.

Because the edge points split the current polygon, each of the 3 neighboring polygons must be split into two polygons as well so that there is no gap (Figure 10). The graphics loop therefore draws 6 polygons in place of the original, 2 polygons in place of each neighboring polygon, and then draws all of the rest of the polygons.

The graphical material effects are accomplished by finding the normals for each vertex, an average of all the normals of the polygons containing that vertex. Then the surface is either Phong or Gouraud shaded according to those normals.

Deformations

The base points in the polygonal algorithm can be given dynamics properties to allow the deformation of an object. This can be accomplished in several ways. Vertices can be given dynamics properties as described above in the dynamics section, with a very small time variable. In this way, the vertices move slowly and create a feel similar to clay. Additionally, the vertices can be moved with a simpler method of simply displacing them proportional to the force presented. This however does not allow as much flexibility in modifying the feel of the object.

There are several issues that arise when the vertices are allowed to move. First, there must be some constraints applied to the vertex movements so that a polygon does not collapse on itself. This can be done by maintaining a minimum distance for a side on a polygon, or by allow the vertices to move in only a specified direction along a line.

An additional problem comes from the oct-tree based culling which allows real time collision detection. If vertices are allowed to move then the pre-processed tree structure, that describes which polygon should be checked for collisions given the cursor is in a specific leaf node, can be made invalid.

Results

The ActivePolygon algorithm was tested on a Dual PIII 800MHZ computer with 512MB of memory, a Wildcat 4210 graphics card, and a Phantom desktop haptic device. The haptics load was obtained using the GHOST 3.1 Haptic Load program. Object 1 has a complex geometry with many areas that are often troublesome. Objects 2 and 3 were the same object and had the same topology, a relatively simple topology, but differed only in their resolution and polygon counts. Object 4 had a large polygon count. There were several aspects of the results that were significant. First, the haptic load averages and peaks were consistent across objects with varying polygon counts. This was expected as the computations are based on a pre-processed data set and are computed locally so the overall size does not affect the local computations. Second, the haptics were stable across varying and complex geometries. Third, the load times increased as the objects had more polygons. This also was expected as there was therefore more data to be preprocessed. A majority of the load time is due to the preprocessing. After the preprocessing is done once, the preprocessed data can be saved with the object, and the load time can be perceptually eliminated. We additionally tested an object with over 1 million polygons. The haptic load peak occurred when initially touching or leaving that object, but maintained a consistent average.

Table 1: ActivePolygon results

Object	Polygons	Load Time (sec.)	Visual FPS	Haptic Load Avg.	Haptic Load Peak
1	5706	1.03	21.33	20%	20%
2	134232	19.54	6.76	15%	20%
3	239694	33.52	3.8	15%	20%
4	1063452	195	0.98	15%	80-100%

Table 2: Comparison with GHOST polygonal renderer

Object	Polygons	Load Time (sec.)	Visual FPS	Haptic Load Avg.	Haptic Load Peak
1	5706	0.9	15	20%	30%
2	134232	11.74	6	85%	100%
3	239694	17.12	6	70%	70%
4	1063452	220		Unstable haptics	

We also compared the ActivePolygon algorithm to the TouchVRML polygonal renderer contained in the GHOST 3.1 API from SensAble Technologies. The listed TouchVRML Visual Frames Per Second are highly qualitative and were obtained by viewing and estimating. In objects 1, 3, and 4 using TouchVRML, the phantom would have force kicks in certain manipulation situations. Objects 3 and 4 had unstable haptics in the TouchVRML application.

Overall, the ActivePolygon algorithm worked very well. Further research is being done to continue to extend its functionality and scope, add modifications such as haptic textures, and integrate it into software applications. The source code for the ActivePolygon algorithm is available in the e-Touch library on the e-Touch web site, www.etouch3d.org.

Acknowledgements

The writers would like to acknowledge Walt Aviles, Sandia National Laboratories, Arthurine Breckenridge, George Davidson, Tom Caudell, and Tom Furness for their help in the development of this work.

References

[1] K. Salisbury, D. Brock, T. Massie, N. Swarup, and C. Zilles, "Haptic Rendering: Programming Touch Interaction with Virtual Objects", ACM Symposium on Interactive 3D Graphics, Monterey, CA, 1995.

[2] T. Massie and K. Salisbury, "The Phantom Haptic Interface: A Device for Probing Virtual Objects", Proceedingsmof the ASME Winter Annual Meeting, Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems, Chicago, IL, 1994.

[3] SensAble Technologies Inc., "The PHANToM" literature from SensAble Technologies.

[4] C. Zilles, J. Salisbury, "A Constraint- based God-object Method for Haptic Display", IEEE/RSJ International Conference on Intelligent Robots and Systems, Human Robot Interaction and Cooperative Robots, 1995.

[5] P. Buttolo, B. Hannaford, B. McNeely, "Introduction to Haptic Simulation", Tutorial 2A, IEEE/VRAIS tutorial notes, March, 1996.